Coding Styles - A Software Engineering Approach

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

N. Shyam Sunder

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

February 1996

CSE-1996-M-SUN-COD

CERTIFICATE

This is to certify that the work contained in the thesis entitled Coding Styles - A Software Engineering Approach by N. Shyam Sunder has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Dr. T. V. Prabhakar,
Associate Professor,
Department of Computer Science & Engineering,
Indian Institute of Technology, Kanpur.

February 1996.

Acknowledgements

First of all I want to thank my father for giving constant encouragement for studying. I like to thank my elder brother for enabling me to take a long leave from my responsibilities towards my parents and younger brothers and sisters.

I am indebted to my guide Dr. T. V. Prabhakar for offering me this thesis in spite of ... I am grateful to him for his guidance, support and the confidence which he has expressed in me. I am thankful to him for his friendly and cordial behavior which made project meetings an enjoyable, and pleasant exercise.

I want thank drc for proofreading this report. I like to thank Ganesan for providing *webtry* which helped me in down loading certain useful documents related to this work.

I am thankful to M.Tech'94 batch for their friendship and sharing many happy moments with me in parties, phata, gappe etc. I express my thanks to drc, davi, pkv, kris, and killi for giving me company on numerous occasions.

Abstract

Good programming style is essential for writing computer programs which can easily be read and understood by human beings. Every part of a stylistic program communicates its intention and functionality clearly to its reader. Conventionally, programming style is a qualitative concept. However, for developing style based tools such as style checker, style analyzer etc., it has to be quantified.

In this work an extensive study of *C* language's coding styles has been done. This work is an attempt to quantify style rules using a specification language, *C-Style Specification Language (C-SSL)*. *C-SSL* is a simple and extendible language. A prototype style checker, *C-Coding Style Checker(C-CSC)*, has also been developed for checking coding style of *C* language programs.

Contents

Chapter 1

Introduction

1.1 Objective

In a software house hundreds of programmers produce millions of lines of code. This code is required to be compiled by number of different compilers and executed on variety of hardware and operating system. Normally a program will be written by few(one or two) programmers but will be read by many persons, many number of times during the different stages of software life cycle. The most important thing is that, normally, code will be maintained by a persons who are not its author. At times even testing will be done by somebody else. Under this kind of scenario it is imperative that every programmer should follow certain rules or guidelines so that produced code will be more readable, portable, maintainable etc.

Normally style rules/guidelines are laid down by some experienced programmers and made available in the form of some document written in some natural language. The task of ensuring that laid down rules are followed by every programmer in his program, is done by some experienced programmer or a software manager usually in a quality control group. He reviews the code, manually checking whether the program conforms to style rules. Sometimes these code-reviews get heated due to the varying tastes of the programmer and the software manager.

The objective of this work is to study the issues pertaining to the programming style rules and tools based on them in particular for the *C* language. Goal is

to develop a mechanism for specifying style rules so that they can be used by various tools. This study reasons out the usefulness of the rules and describes the exceptions if any. A prototype coding style checker has been developed with the aim of automating the process of coding style checking.

1.2 Related work

Publications on programming styles often present either a set of style rules or descriptions about automated style analysis. Pretty printers [WC89] and code formatters concentrate only on the look of a program (only spaces and indentation). A number of style checkers has been written particularly for *Pascal* language. Most of these don't provide much flexibility in specifying the layout. Papers and articles on style analysis have reported certain statistics about the programming style of a program but without any formal basis. These analyzers are basically style grading programs which quantify program characteristics thought to represent programming style and compute a style score as a weighted sum of factors. Importance of these factors remains a controversial issue.

It is surprising to know that hardly any publications are found on automatic style checking of *C* programs in spite of its popularity and extensive usage. It is heard that some of the software houses use some tools for style checking but no information is readily available about them. Some of the significant developments of the past are

lint : A statical checker for C programs[AT 92]

LClint : A more powerful statical checker for C programs than lint [DEH94].

Indent : A C code formatter. It is a GNU's free software[Arc94].

Code–Check : A coding style checker for C programs[Cod90].

The Stanford Ada Style checker : A style checker for Ada programs[Wal91].

ASSESS : Evaluates stylistic factors of Fortran-77 programs[RS86].

GEN++ : An analyzer generator for C++ programs[DE95].

IP-print : A prototype intelligent pretty printer for *Pascal*[WC89].

1.3 Overview

Chapter 2 describes the concept of programming style.
Chapter 3 gives a detailed account of various *C* coding style rules.
Chapter 4 introduces a language for specifying style rules of *C* language.
Chapter 5 describes a prototype coding style checker for *C* language.
Chapter 6 concludes the work and suggests the future work that can be taken up in this area.
Appendix A contains detailed grammar of *C-Style Specification Language (C-SSL).*
Appendix B gives a sample style file.
Appendix C gives a sample list style violation warnings generated by *C-Coding Style Checker(C-CSC).*

Chapter 2

Programming Style

The issue of programming style is as old as programming itself. There exists no general agreement on what constitutes a good style. There are several rules/guidelines which are followed by different groups. Several attempts to formally define them proved difficult, and met with a little success. Over a period, one general agreement arrived at is that, it is an elusive concept with intuitive elements, which make it difficult to define and quantify.

2.1 Programming Style

Some of the attempts to define programming style are

> *The elegance of style of a program is sometimes considered a nebulous attribute that is somehow unquantifiable; a programmer has an intuitive feel for a good or a bad program ...[EE85]*

> *Programming style brings to mind the ways that a creative programmer brings maintainability, testability, reliability and efficiency to the coding of a module [J.85].*

Above definitions show that it is a qualitative concept. The desirable qualities of a good program are as follows:

- Readability
- Maintainability
- Portability[1] and
- Correctness

There are certain other features such as

- Reliability
- Execution speed
- Development time
- Extendibility
- Reusability
- Memory size
- Source code size etc.

which are also considered as desirable properties of a programming style. But the importance of some of the features of the latter list has gone down with time and some others are implicitly covered by one or more features of the first list. This means that

> *Programming style is a list of prescriptions, prescribed by experienced programmers, which help in achieving readability, portability, maintainability and avoiding certain programming traps and pitfalls.*

Following are the sample programming style rules:

[1]Source code is said to be portable if it can be compiled and executed on different machines, with the only change being the inclusion of possibly different header files, few alternate source files and the use of different compiler flags. Header files shall contain #defines and typedefs, which may vary from machine to machine. In general a new machine is a different hardware, a different operating system or a different compiler or any combination of these.

- All names shall be meaningful.
- Unconditional jumps shall be avoided.

The rules which facilitate writing portable, maintainable code are free from debate . On the other hand those pertaining to other two are arguable and lack general acceptance. This is due to the inherent subjectivity of the concept of readability itself. A piece of code may be considered readable by one group of programmers whereas some other group may not feel so.

```
if (account_balance>0) {
    NoofTrans++;
    process_trans(TransId);
    account_balance -= debit_amount;
}
```

Listing - 2.1a.

```
if (acc_bal > 0.0 )
{
    tcount++;
    process_trans (trans_id);
    acc_bal -= deb_amt;
}
```

Listing - 2.1b.

Different programmers may have different views on the listings 2.1a. and 2.1b shown above. It is difficult to rank one above the other and give reasons for doing so. The most difficult is to make others accept the ranking. This just gives an idea about the subjectivity of the issue.

2.2 Coding Style

Normally programming style and coding style are not considered different from each other. For the sake of better understanding of the under lying concepts it is advantageous to make distinction between them. Programming style is primarily

determined by the class of language or the programming model under consideration. For example:

- Block Structured
- Object Oriented
- Functional
- Logic etc.

Concepts relating to programming style such as naming, program layout, robustness etc. can be discussed in a manner independent of any language, but certain other concepts like inheritance, backtracking, information hiding, side effects etc. don't make much sense for one or more class of languages.

Any two languages belonging to the same class of language differ in syntax, semantics and features provided by them. For example external variables, static data, void pointer etc. are irrelevant for *pascal*, whereas any discussion on *C* coding styles is incomplete without them. This means that we can't have the same set of style rules even for all the languages of a class. Hence we can define coding style as

> *A set of rules, suggested by some experienced programmers for a language, which help in achieving readability, portability, maintainability and avoiding programming and language's traps and pitfalls.*

Another way programming style and coding style can be defined as:

> *All rules which help in achieving readability, portability, maintainability, and avoiding programming traps and pitfalls constitute programming style.*
>
> *Coding style of a programming language is a set of programming style rules which can be defined in terms of syntax and semantics of the language.*

Following are the sample *C* coding style rules:

- Sub-words in an identifier name shall be separated either by an underscore or a capital letter as shown below:

  identifier_name or *IdentifierName*

- Identifier names shall not start with an underscore.
- Identifier names of #defined constants and enumerated constants shall be in upper case.
- Use of *goto* shall be avoided.
- *break* and *continue* statement used inside *for/while/do* loop shall be commented.

Defining coding style in this manner facilitates the addressing of specific features of a language to the fullest extent. It also provides a basis for the development of automatic tools based on them.

2.3 Coding Standard

For any programming problem there can be a number of equally correct programs providing its solution, but differing in some way from each other. This happens even when all of them are based on the same algorithm, and satisfy style rules. This is because of choices existing in the style rules and some may not follow certain rules due to the subjectivity of the rules. Programs can be formatted in more than one way. There are many naming conventions, statement orderings, commenting styles etc. If every programmer selects style options as per his taste then one will face more problems in understanding somebody else's program. This finally gets reflected as revised project schedules and higher maintenance cost. These inconsistencies also devalue the entire group's effort. It is unfortunate that these inconsistencies are called creativity by some groups.

Now we can define coding standard as a set of cohesive coding style rules with the selected options if they exist as per the requirements and taste of a group. Here

group means a group of programmers working together on a project etc. This implies that coding standard may differ from group to group. In practice there exist some exceptions to the style rules. A coding standard document should mention them. It should also give the rationale behind them.

Following are the sample C coding standard rules:

- Variable names shall be in lower case only. Sub-words shall be separated by an underscore.
- Identifier names shall not start with an underscore.
- Identifier names of #defined constants and enumerated constants shall be in upper case.
- *goto* statement shall not be used.
- *break* and *continue* statement used inside for/while/do loop shall be commented.

Coding standards can be broadly classified into two classes [OC88].

Typographic: covers include level, indentation, line length, placement of comments, placement of blank lines, use of embedded spaces, identifier length, module length, and format for type and data declarations.

Structural: covers modularity, use of labels and gotos, use of named constants, use of include files, use of literals, method of type and data definitions, use of library functions level of nesting, control flow, information flow, operator and operand usage, and factors related to program complexity.

With the development of formatting tools like Indent[Arc94] most of the typographic factors are no longer considered crucial. One can easily reformat program as per his flavor using *Indent* or tools similar to it. This makes other factors more important because of the difficulties in automatically checking or achieving them.

2.4 Applications of Coding Styles

Some of the useful tools which can be developed based on coding styles are:

Style Checker: Checking programs for style violations.

Code Converter: Converting a program to another program which satisfies given coding standard.

Quality Workbench: An interactive tool for inspecting and editing programs. Given a style rule it shows parts of the program, on which given rule is applicable.

Style Based Editor: Checking style violations as code is being edited/typed, and applying style transformations wherever possible, otherwise giving style violation warnings.

Style Analyzer: Given a set of style rules and associated weights computing a
ased on this score a program may be categorized

Chapter 3

C Coding styles

C is the one of the most popular and widely used programming language. Most of the system software is written in C or lately in its daughter C++ is being used. In the words of Andrew Koenig[Koe89]

> *The C language is like a carving knife: simple, sharp and extremely useful in skilled hands. Like any sharp tool C can injure people who don't know how to handle it.*

C is full of traps and pitfalls. There are a number of ways one can unwittingly insert obscure bugs. A novice programmer may end up spending hours together debugging his code, which ultimately ends up in finding a simple typographical mistake. These are the reason for which we require good style rules and style based tools.

Style rules can be classified into following classes

- Syntactic Rules
  - Data definition and declaration rules
  - Function definition and declaration rules
  - Statement rules
- Internal documentation rules

- Naming rules
- Layout rules
  - Spacing rules
  - Indentation rules
- General practices
- Preprocessor rules

Rest of the chapter describes an extensive set of style rules. Many of these rules can found in [Koe89], [Tas78], [AD90], [Pik90], [HS89], [FSF94], and [Ous89]. This list also includes the rules which can't be defined in terms syntax and semantics of the *C* language, at present.

3.1 Syntactic Rules

3.1.1 Data definition and declaration rules

- Identifier declarations/definitions shall not span across lines. Preferably only one identifier shall be declared/defined in a single statement. Multiple local identifiers may be declared/defined in a same statement provided inclosed block's size is not more than a few lines.

```
int  foo_1,                /* DISCOURAGED */
     foo_2;

int  foo_1;                /* RECOMMENDED */
int  foo_2;                /* RECOMMENDED */
```

Option: *Yes/Don't care* **Benefits:** *Readability, Encourages documentation*

- Initialized variable definitions shall not be combined with uninitialized variable definitions in a same definition statement. There shall be only one initialized variable definition in a single statement.

Option: *Yes/Don't care* **Benefits:***Readability, Encourages documentation*

- Most significant dimension of an externally declared array variable or an argument shall not be specified. In case of array declared as argument, size of the array shall be passed as an argument. This facilitate passing of different size array on different invocations of the function.

```
/* DISCOURAGED */
extern int foo_array[80][100]

char *
foo_func (int array_arg[80][100])
{
    ...
}

/* RECOMMENDED */
#define YDIM_SIZE   100
extern int foo_array[][YDIM_SIZE];

char *
foo_func (int array_arg[][YDIM_SIZE], int xdim_size)
{
    ...
}
```

Option: *Yes/Don't care* **Benefits:***Modifiability, Reusability*

- Most significant dimensions of an initialized array shall not be specified when array is fully initialized. It avoids change in the dimension when number of elements in the initialization list are modified.

  Option: *Yes/Don't care* **Benefits:***Modifiability*

- Static identifiers shall be highlighted by initializing them.

  Option: *Yes/No/Don't care* **Benefits:***Readability, Maintainability*

- All identifiers whose initial value is important shall be initialized at the time of definition. This probably saves few core dumps, which often occur due to uninitialized data.

  Option: *Yes/Don't care* **Benefits:***Reduces errors*

- All related identifiers shall be defined/declared together, and a comment shall precede their declaration/definition.

  Option: *Yes/Don't care* **Benefits:***Readability, Maintainability*

- All statements of a particular type shall be declared/defined together.

  Option: *Yes/Don't care* **Benefits:***Readability, Maintainability, tractability*

- Type definition for *struct/union/enum* shall be combined with the corresponding definition. Type definition shall be present for corresponding pointer type as well.

  Option: *Yes/No/Don't care* **Benefits:***Readability, Maintainability*

- Initialized arrays shall not be defined in a local scope. If possible they shall be declared static.

  Option: *Yes/Don't care* **Benefits:***Readability, Maintainability, Clarity*

- Expressions defining the array dimensions shall be simple #defined expressions.

```
#define  HASH_TABLE_SIZE   100
int  hash_table[HASH_TABLE_SIZE + 1];
```

  Option: *Yes/Don't care* **Benefits:***Readability, Maintainability*

- Type shall be specified for every identifier. Default int type feature shall not be used.

  Option: *Yes/Don't care* **Benefits:***Readability, Maintainability*

- No assumptions shall be made about the offsets of the fields within a structure. Due to alignment, field offset may vary from machine to machine.

  Option: *Yes/Don't care* **Benefits:***Portability*

- *struct, union,* and *enum* shall be defined in a separate file.

  Option: *Yes/Don't care* **Benefits:***Maintainability*

3.1.2 Function definition and declaration rules

- Function arguments in function definition/declaration shall not be cluttered within a single line.

```
/* DISCOURAGED */
int
foo_func (int arg1, char arg2, float arg3)
{
    ...
}

/* RECOMMENDED */
int
foo_func (int arg1,              /* What is arg1 */
          char arg2,             /* What is arg2 */
          float arg3)            /* What is arg3 */
{
    ...
}
```

Option: *Yes/Don't care* **Benefits:** *Readability, Maintainability, Documentability*

- Every function shall be defined as shown below. This provides a marker for function definitions.

```
#define  FUNCTION

FUNCTION
int
foo_function(void)
{
    ...
}
```

Option: *Yes/Don't care* **Benefits:***Readability, searchability*

- Function prototypes shall be declared for all the functions and they shall be defined with return type, and complete typed list of arguments.

  Option: *Yes/Don't care* **Benefits:***Readability, reduces interfacing errors*

- External declarations shall not be put inside a function. Normally many functions defined in a file will use same external function/variable. Therefore all these external declarations shall be placed together. On the other hand if size of the function is less than 25 lines(one screen) then declaring externs locally will enhance readability at the cost of maintainability.

  Option: *Yes/Don't care* **Benefits:***Readability, Maintainability*

- *struct/union/enum* shall not be defined inside function or in local scope.

  Option: *Yes/Don't care* **Benefits:***Maintainability, Consistent layout*

- Identifiers of type *float/double* shall not be used with equality operator, or compared for equality.

  Option: *Yes/Don't care* **Benefits:***Portability, Reliability*

- Identifier names shall not be redefined inside local scope.

  Option: *Yes/Don't care* **Benefits:***Readability, Maintainability, Documentability*

- Functions shall not have large number of arguments.

  Option: *Number of args* **Benefits:***Readability, Maintainability*

- *main* shall return an operating system defined value. It shall terminate through *exit* system call. This ensures the execution of *at-exit* functions and checking of program's return status by shell scripts.

  Option: *Yes/Don't care* **Benefits:***Reusability, Reduces errors, Maintainability*

- Every function shall have preferably one exit point.

  Option: *Yes/Don't care* **Benefits:***Readability, Maintainability*

- All functions returning pointer shall return *NULL* on failure.

  Option: *Yes/Don't care* **Benefits:***Reduces errors, Maintainability*

- Function shall always return an integer value indicating it's success or failure. However function returning a pointer shall return *NULL* to indicate failure.

  Option: *Yes/Don't care* **Benefits:***Maintainability, Reusability*

- *envp* command line argument shall not be used. It may not available on some machines.

  Option: *Yes/Don't care* **Benefits:***Portability*

- Command line argument, *argv[0]* shall not be assumed to give the complete file name. On some machines it may contain only relative file name.
  Benefits : Portability

- Function shall not return structure. Different compilers may use different conventions for passing and returning structures. This may causes a problem when libraries return structure values to a code compiled with a different compiler.

  Option: *Yes/Don't care* **Benefits:***Portability*

- Size of a function shall not be more than 50 lines.

  Option:*Number of Lines* **Benefits:***Maintainability*

3.1.3 Statement rules

- Nested if statements shall be braced completely to indicate the nesting.

```
/* DISCOURAGED */
if (...)
    if (...)
        ...
    else
        ...
```

```
/* RECOMMENDED */
if (...)
{
    if (...)
        ...
    else
        ...
}
```

  Option: *Yes/Don't care* **Benefits:***Readability*

- Assignment operators shall not be used inside a conditional expression. But in the following case doing so seems better.

```
while ((ch = getchar ()) != EOF )
{
    ...
}
```

Option: *Yes/Don't care* **Benefits:***Readability, Maintainability*

- Embedded assignments shall not be used.

  Option: *Yes/Don't care* **Benefits:***Readability*

- Switch shall have a break statement for every case. When fall through feature is required then a comment shall be put after the last statement to highlight the reason behind it.

  Option: *Yes/Don't care* **Benefits:***Maintainablity*

- Switch shall have default case and it shall be put at the last.

  Option: *Yes/Don't care* **Benefits:***Maintainability, error detection*

- Conditional expressions shall not contain auto increment '++' and auto decrement '- -' operators.

  Option: *Yes/Don't care* **Benefits:***Readability, Maintainability*

- Conditional expressions shall not contain function calls. In the following case doing so seems better.

```
while ((ch = getchar ()) != EOF )
{
    ...
}
```

  Option: *Yes/Don't care* **Benefits:***Readability, Maintainability*

- Index expression shall be a simple expression. It shall not have a assignment operator or a function call.

  Option: *Yes/Don't care* **Benefits:***Readability*

- Pointer type identifiers shall not be used with other than binary '+', '-' and relational operators. Only pointers of the same type shall be compared.

  Option: *Yes/Don't care* **Benefits:** *Portability, Maintainability, Readability*

- Actual arguments of a macro call shall not contain any auto operator. when a argument is used more than once in the replacement text, side effects may take place, which may result in obscure bugs.

  Option: *Yes/Don't care* **Benefits:** *Reduces errors, Readability*

- Conditional expression of a ternary operator shall be in parenthesis and nested ternary operators shall not not be used.

  Option: *Yes/Don't care* **Benefits:** *Readability, Maintainability*

- All type conversions shall be explicitly typecasted. This makes the programmers intent clear.

  Option: *Yes/Don't care* **Benefits:** *Maintainability*

- *NULL* shall never be used as end of string(EOS). EOS shall be #defined as

  ```
  #define  EOS    '\0'
  ```

  Option: *Yes/Don't care* **Benefits:** *Readability, Maintainability, Portability*

- *TRUE* and *FALSE* shall be used only in the boolean context. They shall be defined as follows:

  ```
  typedef enum
  {
      FALSE = 0, TRUE = 1
  } boolean;
  ```

  Option: *Yes/Don't care* **Benefits:** *Portability, Readability*

- Every system call and call to memory allocation functions shall be checked for a possible failure.

  Option: *Yes/Don't care* **Benefits:** *Robustness, Maintainability*

- No assumptions shall be made about the evaluation order. In the following code we can't guarantee whether a[0] or a[1] will be set.

```
i = 0;
a[i] = b[i++];
```

Option: *Yes/Don't care* **Benefits:***Portability*

- *unsigned* variables and char type variables shall not be compared with EOF. On some machines default char type may be unsigned. In such a case following code will be an infinite loop.

```
char  ch;

while ((ch = getchar ()) != EOF )
{
    ...
}
```

Option: *Yes/Don't care* **Benefits:***Portability*

- No assumptions shall be made about the word size. Following code may not have a desired effect of clearing rightmost 3 bits.

```
int  num;

num = ...
num &= 0177770;
```

Option: *Yes/Don't care* **Benefits:***Portability*

- Values of a larger range shall not be assigned to an identifier having a smaller range.

Option: *Yes/Don't care* **Benefits:***Portability*

- Pointers of non basic data type shall not be cast to a pointer of of other type.

Option: *Yes/Don't care* **Benefits:***Portability*

- String constants shall not be modified. Following code may have an unexpected behavior.

```
string = ''/dev/tty??'';
strcpy (&string[8], ttychars);
```

Option: *Yes/Don't care* **Benefits:***Portability*

- Expressions shall not contain more than one function calls. In the following code if both functions modify the same variable then program may behave differently on different machines, due to the difference in order of evaluation.

```
/* foo_1 & foo_2 changes value */
result = foo_1 () + foo_2 () + value;
```

Option: *Yes/Don't care* **Benefits:***Portability*

- Negative numbers shall not be used in integer division. This is not defined in the language.

Option: *Yes/Don't care* **Benefits:***Portability*

- There shall be at the most one statement in any line.

Option: *Yes/Don't care* **Benefits:***Readability*

- *goto* shall be avoided as far as possible.

Option: *Yes/Don't care* **Benefits:***Readability, Maintainability*

- Normally only decimal constants shall be used. It is preferable to use hexadecimal constants in bitwise operations.

Option: *Yes/Don't care* **Benefits:***Readability*

- Conditional test shall not default for non-zero.

```
/* DISCOURAGED */
if ( foo() )
    ...
```

```
/* RECOMMENDED */
if ( foo() != FAIL )
    ...
```

3.2 Internal Documentation Rules

Internal documentation or comments are an essential part of any long life program. It is required to describe the relationship between different code segments of a program. Comments shall be short and to the point. Comments and code shall not contradict each other. Good comments enhances readability and maintainability. Comments are of three types.

Header comment: Used for a file header and function header.

Paragraph Comment: Used before a block of code, describing a data structure, algorithm etc.

One-line Comment: Used for describing an important line of code.

- Comments shall be put at the following places.
  - User include files, describing its purpose.
  - End of file.
  - File header. It shall contain name of the authors, date of last modification, synopsis of file contents, purpose of file, dependencies, modification history etc.
  - Fall through of a switch statement.
  - At the start of every block of code. This comment shall establish a clear mapping between the algorithm and the code.
  - *static* identifiers.
  - Global identifiers.
  - *const* identifiers.
  - Function arguments definition.
  - Non portable code segments.
  - Bit field definitions.
  - *goto* statement.

- Label statement.
- Hacked code segments.
- *struct*, *enum*, and *union* definitions.
- type definitions.
- #define constant definitions.

- Function header. It shall contain purpose and meaning of arguments. description of return value, dependencies, precautions, assumptions, side effects etc.

- If a function is more than 25 lines (one screen) then it shall have a end of function comment as shown below.

```
int
foo_func (...)
{
    ...
} /* End of function foo_func (...) */
```

- If the body of *if, else, switch, case, for, while, and do* is more than 25 lines (one screen) then there shall be a comment at the end of the body as shown below.

```
if (condition)
{
    ...
} /* End of if ( condition)  */
else
{
    ...
} /* End of else ( condition) */

switch ( expression)
{
```

```
 case value :
{
     ...
} /* End of case ( value)  */
...
} /* End of switch ( expression)  */

while ( condition)
{
    ...
} /* End of while ( condition)  */

do /* Begin of do ( condition)  */
{
    ...
} while ( condition);

for (...; ( condition); ...)
{
    ...
}/* End of for ( condition)  */
```

3.3 Naming Rules

Identifier names shall be mnemonic names, clearly indicating the obvious purpose for which they are used. They shall have some relation with the entity/concept they represents in a program. This relation ship shall be followed consistently in all the names used in a program. Well chosen names can make a program self documented.

- File name shall be in 11.3 format, where 11 is the length of full name and 3 is the length of extension. It shall be containing only letters, digits, underscore and dot only.

Option: *Yes/Don't care* **Benefits:***Portability*

- First 31 characters of all identifiers shall be unique in a compilation unit.

  Option: *Yes/Don't care* **Benefits:***Portability*

- First 6 characters of all global identifiers shall be unique. This is due the limitations of *linker* on some machines.

  Option: *Yes/Don't care* **Benefits:***Portability*

- Following file name extensions shall be used.

file-type	*extension*
source	.c
include	.h
assembly	.asm
assembly	.inc
include	
data	.dat

  Option: *Yes/Don't care* **Benefits:***Portability*

- #defined constants, constant identifiers and enumerated constants shall be in upper case.

  Option: *Yes/Don't care* **Benefits:***Readability*

- Identifier names shall contain a prefix indicating project, module, type, and scope.

  Option: *Yes/Don't care* **Benefits:***Readability, Maintainability*

- Same name shall not be used for multiple purposes in different parts of a program.

  Option: *Yes/Don't care* **Benefits:***Maintainability, Reduces errors*

- Escape character constants shall be #defined

  Option: *Yes/Don't care* **Benefits:***Readability*

- Function name shall have a prefix indicating return type.

Option: *Yes/Don't care* **Benefits:***Maintainability*

- Argument names shall have a prefix indicating their usage as input, output or both.

  Option: *Yes/Don't care* **Benefits:***Readability, Maintainability*

- Identifiers names shall not start with a underscore. Doing so may create conflicts with system defined names.

  Option: *Yes/Don't care* **Benefits:***Maintainability, Portability*

3.4 Layout rules

3.4.1 Spacing rules

- There shall be a blank line before function body separating it from function name and argument definitions.

  Option:*Number of lines* **Benefits:***Readability, Encourages documentation*

- There shall be two blank lines separating data definitions/declarations from statements.

  Option:*Number of lines* **Benefits:***Readability*

- Code shall be organized in blocks. There shall be two blanks between two consecutive blocks.

  Option:*Number of lines* **Benefits:***Readability*

- There shall be no space around primary operators `'->'` and `'.'`.

  Option: *Yes/Don't care* **Benefits:***Readability*

- There shall be no space between unary operator and it's operand.

  Option: *Yes/Don't care* **Benefits:***Readability*

- Binary and Assignment operators shall have one space around them.

  Option: *Yes/Don't care* **Benefits:***Readability*

- There shall be at least one space or a newline after ',' and ';'.

  Option: *Yes/Don't care* **Benefits:***Readability*

- There shall be four/five lines between two consecutive function definitions.

  Option:*Number of lines* **Benefits:***Readability*

3.4.2 Indentation rules

- A *if* statement in the body of else shall be formatted as follows.

```
if (...)
    ...
else if (...)
    ...
```

```
if (...)
    ...
else
{
    if (...)
        ...
}
```

  Option:*Indent type* **Benefits:***Readability*

- All block statements *if*, *switch*, *for*, *while*, *do*, and *body of case* shall be braced completely.
  Option:*Indent type* **Benefits:***Readability, consistent layout*
- Label shall start at column no 1 or at the next higher indentation level. It should be on a line by itself.
  Option:*Indent type* **Benefits:***Readability*
- Lengthy expressions shall be spilt onto multiple lines. They shall be parenthesize appropriately to indicate the nesting, and desired operator precedence.

  Option:*Indent type* **Benefits:***Readability*
- If *for* loop's expressions (initialization, conditional and iteration expressions) are lengthy then each shall start on a separate line.
  Option: *Yes/Don't care* **Benefits:***Readability*
- Initialized data shall be indented properly to indicate it's structure.
  Option:*Indent type* **Benefits:***Readability*
- Size of indentation shall be 2 to 4 spaces.
  Option:*Number of spaces* **Benefits:***Readability*
- Only braces defining a function shall be on column 1.
  Option: *Yes/Don't care* **Benefits:***Readability, Maintainability*
- Null body of a statement shall be indented properly.
  Option:*Indent type* **Benefits:***Readability, Maintainability*

3.5 General Practices

- Indexing operator shall be used only with array type identifiers and dereference operator shall not be used for accessing array elements.

  Option: *Yes/Don't care* **Benefits:** *Readability, clear and non cryptic code*

- In general hard coded magic numbers shall not be used. They shall be replaced by enumerated constants(preferably) or #defined constants. Depending on the context trivial constants such as 0, 1 etc. can be used as it is.

  Option: *Yes/Don't care* **Benefits:** *Readability, Maintainability*

- Literal constants shall indicate the desired type. Only upper case letters shall be used in them.

```
long   long_var = 1L;                    /* RECOMMENDED */
float  float_var = 0.0;                  /* RECOMMENDED */

long   long_var = 1l;                    /* DISCOURAGED */
long   long_var = 1;                     /* DISCOURAGED */
float  float_var = 0;                    /* DISCOURAGED */
```

  Option: *Yes/Don't care* **Benefits:** *Readability*

- Include files shall not contain data definitions.

  Option: *Yes/Don't care* **Benefits:** *Reduces errors, Maintainability*

- Limits on the length or size of data structures, filenames, etc. shall be avoided.

  Benefits : Robustness, Reusability

- Utilities processing files shall not drop null or any other non printable characters.

  Benefits : Robustness, Reduces errors

- Standard library function *free* shall be expected to modify the contents of the freed block. No reference shall be made to the freed block.

  Option: *Yes/Don't care* **Benefits:** *Reduces errors, Portability*

- Data which will not modified by the program shall be set using initialized definitions. It should be of declared as *const* type and *static* if possible.

  Option: *Yes/Don't care* **Benefits:** *Maintainability, Reduces errors*

- Non portable code shall be identified and organized in separate files.
  Benefits : Portability
- Standard library functions shall be used instead of equivalent user defined or system specific functions.
  Benefits : Portability
- Only ANSI-C shall be used as far as possible. Code non conforming to ANSI shall be organized in separate files.
  Benefits : Portability
- Assumption about ASCII shall be avoided or char shall not be assumed to be of 8 bits only.
  Benefits : Portability
- Arithmetic operations shall not be replaced by equivalent shift operations.
  Benefits : Portability
- No assumptions shall be made about parameter passing mechanism (size of pointers, parameter evaluation order, and size of parameters). In the following code *ch* may not return 2nd read character. This may happen due to the direction in which stack grows, or absence of a stack, or widening of arguments, or any combination of these.

```
char
foo (char ch1, char ch2, char ch3)
{
    char ch4 = &(c1 + 1);
    return (ch4);
}

ch = foo (getchar (), getchar ());
```

  Benefits : Portability

- Backslash newline pairs shall be used only in macro function definitions, and macro constant definitions.

  Option: *Yes/Don't care* **Benefits:***Readability*

- Strings shall not be split across lines using a backslash.

  Option: *Yes/Don't care* **Benefits:***Readability*

- Size of a block of code shall not be more than 10 lines.

  Option:*Number of lines* **Benefits:***Readability*

- All global identifiers shall be defined in a single file.

  Option: *Yes/Don't care* **Benefits:***Maintainability*

- All identifiers shall be initialized prior to being passed as an argument to a function.

  Option: *Yes/Don't care* **Benefits:***Reduces errors*

- All file inclusions shall be idempotent. Following shall be put in every include file to avoid multiple inclusions.

```
#ifndef _FILE_NAME_H_
#define _FILE_NAME_H_
... /* Body of a include file */
#endif /* End of filename.h */
```

- An open file shall be closed as soon as it is no longer required.

  Option: *Yes/Don't care* **Benefits:***Maintainability*

- Dynamically allocated memory shall be freed as soon it's usage is over.

  Option: *Yes/Don't care* **Benefits:***Maintainability*

- Program shall not contain any compiler or lint warnings

  Option: *Yes/Don't care* **Benefits:***Maintainability*

- Size of source file shall not be more than 1000 lines.

  Option:*Number of lines* **Benefits:***Readability, Maintainability*

3.6 Preprocessor rules

- #define shall not be used for emulating typedef. If we have

  ```
  /* DISCOURAGED */
  #define foo_rec struct foo_rec
  ```

  then subsequent redefinition will produce and warning where as redefinition of typedef will produce an error.

  Option: *Yes/Don't care* **Benefits:** *Reduces errors*

- Parenthesis shall be put around each of the macro arguments in the replacement text of the macro definition. This ensures that arguments are evaluated as desired, provided there are no side effects.

  Option: *Yes/Don't care* **Benefits:** *Reduces errors, maintainability*

- Macro definition shall not start with assignment operator.

  Option: *Yes/Don't care* **Benefits:** *Reduces errors*

- Macro definition shall not end with a ';'. This will avoid redundant null statements.

  Option: *Yes/Don't care* **Benefits:** *Reduces errors*

- Replacement text of a macro constant/function shall be separated from the macro name by two or more number of spaces.

  Option: *Indent type* **Benefits:** *Reduces errors*

- Macro functions shall not have large number of arguments.

  Option: *Number of argume* **Benefits:** *Readability, Maintainability*

- Macros shall not be redefined

  Option: *Yes/Don't care* **Benefits:** *Maintainability*

Chapter 4

C-Style Specification Language

C-style specification language*(C-SSL)* is a language for specifying *C* coding standard. In order to automate any process which requires coding styles, style rules have to be specified in some formal machine understandable manner. In a software house style rules are set by some experienced programmer(system manager), and followed by all the programmers. It is changed infrequently and marginally over a long period. Considering the above factor a high level approach is required for style specification.

4.1 Design Goals

Simple Syntax : Should not have large number of operators and constructs.

Obvious Semantics: Specified rules should be self explanatory.

Concept Based : Should be based on concepts related to programming style and the *C* language.

Extendible : Should be able to add new style rules in the existing language structure.

4.2 Syntax and Semantics

Coding standard specified in *C-SSL* is a sequence of *C-SSL* statements. Each statement describes a constraint on a concept or a language construct in an environment. Structure of grammar is shown in the following productions specified in BNF[1].

```
coding_standard : statement_list
statement       : environment { rule_list }
rule            : concept = constraint ;
                | group = { rule_list } ;
```

Following is the list of environments:

- *code_file :* A unit of compilation
- *pp :* Preprocessor
- *func_def :* Function definition
- *func_decl :* Function declaration
- *statement :* Executable statement
- *comment :*
- *constant :* Numerical, character and string constants
- *expression :*
- *ddd :* Data declaration and definition
- *op :* Operator
- *name :* Identifier names

[1]Backus Normal Form

Groups are basically provided to group certain set of rules so that they can be specified by a single statement. Following is the list of groups:

- *comment :*
- *size :*
- *position :*
- *ddd :*
- *macro_const :* #defined preprocessor constant
- *macro_func :* #defined preprocessor function
- *include :* File inclusion directive.
- *pp_if : #if, #ifndef, #ifdef*
- *hash :* Preprocessor's '#' character.
- *if : if* statement
- *switch : switch* statement
- *loop : while*, *for* and *do* statement
- *index_expr :* Index expression.
- *logic_expr :* Logic or conditional expression
- *call_expr :* Call expression
- *block :* Block comment.
- *struct : struct/union*
- *array :*
- *enum :*

Following is the list of constraints.[2].

- *mask :* A option value Yes/No/Don't care
- *number :* An unsigned integer used for specifying sizes, number of lines, number of spaces etc.
- *string :*
- *type_list :* List of data types. Considered types are *void, char, uchar (unsigned char), short, ushort (unsigned short), int, uint (unsigned int), long, ulong (unsigned long), float, double, struct, union, array, enum, pointer, and usertype (typedefed type).*
- *stmnt_list :* List of statements. *sys_include, user_include, macro_func, macro_const, typedef, struct (definition), enum(definition), extern_var, extern_static_func, static_var, global_var, func_def, static_func_def* are the type of statements.
- *indent_option* Used for specifying type of indentation. It consists of *pos_type* and *number*. *pos_type* is used to indicate the relative position. It can be one of the following:
  - *spaces :*
  - *newline :*
  - *newline_spaces :* Last + spaces
  - *align :* Align with the current indentation.
  - *align_spaces :* Last + spaces
  - *arg_start :* Argument start position
  - *arg_start_spaces :* Last + spaces
  - *index_start :* Index expression start position.
  - *index_start_spaces :* Last + spaces

[2]Upper case names are the tokens of *C-SSL*

- *call_start* : Function call's argument start position.
- *call_start_spaces* : Last + spaces

- *op_list* : List of operators.
- *opname_list* : List of operator names
- *op_entry* : It consist of operator, *number*(prefix space) and *number* (suffix spaces)

Complete grammar of *C-SSL* along with descriptions of concepts and constraints is given in the Appendix A.

Chapter 5

C–Coding Style Checker

Style checking of *C* programs is a process which extends the standard grammar to include spaces, newlines, textual position of sentences, and placement of comments and restricts some of the sentences of the language. C - Coding Style Checker *(C-CSC)* is a prototype style checker for the *C* language programs. It uses style rules specified in *C-SSL*. Figure 5.1 shows the process of style checking.

5.1 Implementation

C-CSC performs two passes over the given input program.

Pass-I Preprocessing

- Builds macro definitions.
- Filters conditional compilations statement. Part of the code which will not be compiled is put inside comments. This way textual positions of statements remains unchanged.
- Macros are not expanded.

Pass-II Parsing and style checking.

- Builds symbol table
- On recognizing productions makes a call to style checking functions.

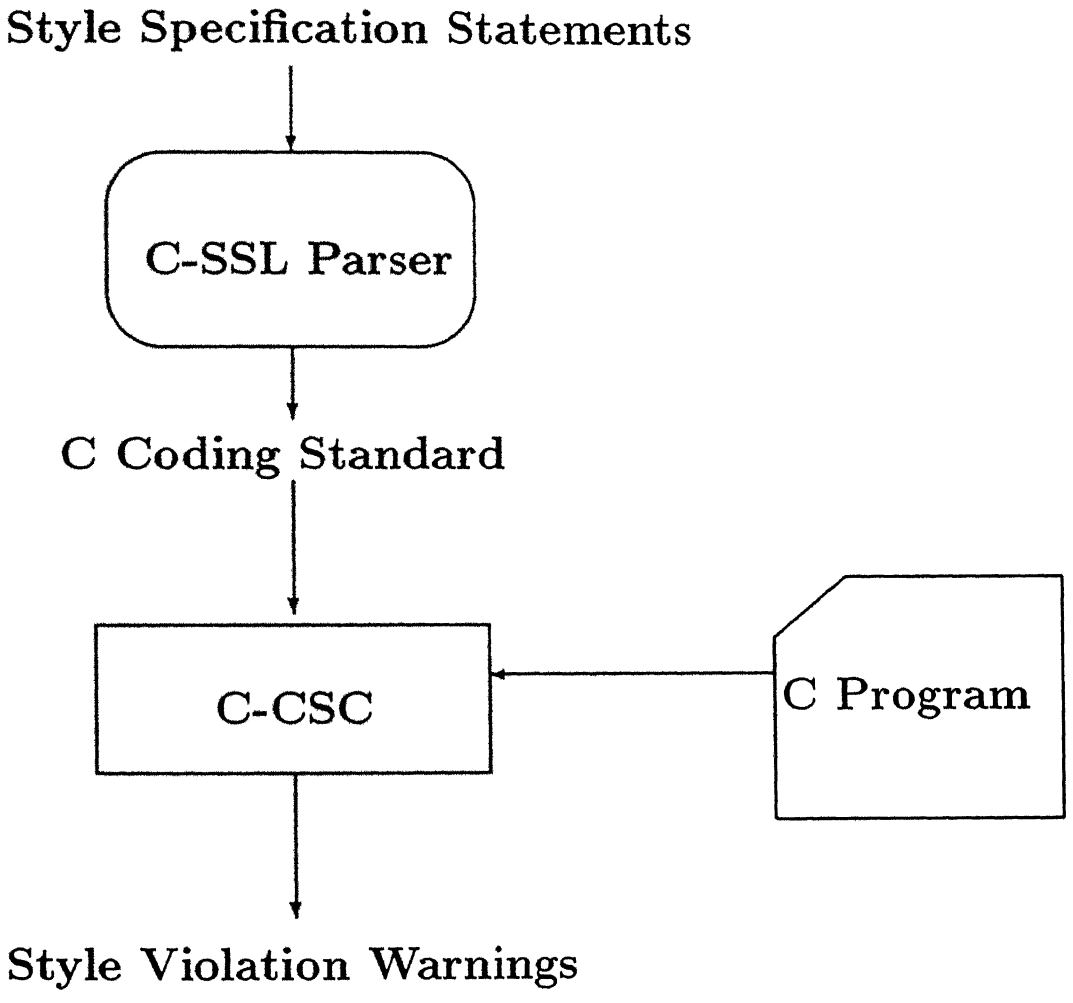

Figure 5.1: Style Checking Process

Style checking functions perform necessary checking depending on a rule if that rule is set in the given coding standard. On detecting style violations appropriate violation messages are generated.

5.2 Usage

C-CSC accepts two command line arguments, a coding standard file and a *C* program file.

```
<prompt:> ccsc <Coding Standard File> <Program File>
```

Chapter 6

Conclusions

6.1 Conclusions

This work is an attempt to develop coding style based tools. Although *C-SSL* and *C-CSC* are meant exclusively for the *C* language but the methodology and the concepts discussed in earlier chapters can be applied for other languages as well. *C-SSL* presents an easy way of specifying coding styles. Using it we can specify most of the popularly used style rules. It can easily be extended to incorporate other rules which are presently not covered. Simple and readable statements of *C-SSL* can serve as an on-line reference in addition to the voluminous documents. *C-CSC* is a prototype style checker. It can check a subset of GNU coding standard. It doesn't perform any kind of data flow analysis. Very few style rules require data flow analysis, but without them style checking can't be complete.

6.2 Exceptions

- *C-CSC* assumes that input program is free from compilation errors and is written in ANSI C.
- Backslash newline pairs shall not be present within identifier names, keywords, constants, and '/*'.

- Old style argument declarations not supported.
- Include files are searched in /usr/include and current working directory only.
- Trigraph sequences, '#' and '##' operators are not handled.

6.3 Future Work

Certain programming style rules are qualitative. They can't be checked algorithmically with presently available techniques. Further work can be done to define these rules in terms of language's syntax and semantics, so that they can be checked automatically.

We can check the presence or absence of a comment. Interpreting the meaning of comment text, or its relevance to the code is a natural language understanding problem. But certain approximate techniques can be developed based on the presence of identifier names and/or corresponding entity names represented by identifiers in a program, in comment text. *C-CSC* checks only a single program file at a time. A good style checker should understand the concept of project or module. This is required for checking consistency of coding styles in all the files of a project/module.

It is desirable that consistent naming conventions shall be followed. There will be a number ways an identifier can be named meaningfully. Further studies can be made to understand naming conventions. It may be possible to establish some relations between identifier names and object names so that their meaningfulness can be checked by automatic tools.

Appendix A

C-SSL Grammar

Token names are in upper case.

```
style : style_rule_list
        ;
mask:
          YES
        | NO
        ;
case_option:
          UPPER
        | LOWER
        | MIX
        ;

        // Relative position with respect to the previous token/sentence
pos_type:
          SPACES
        | NEWLINE
        | NEWLINE_SPACES
          // Align with start column of the statement/current indentation
        | ALIGN
        | ALIGN_SPACES
          // Align with start column of arggument s
        | ARGS_START
        | ARGS_START_SPACES
          // Align with start column of index expression
```

```
        | INDEX_START
        | INDEX_START_SPACES
          // Align with start column of call's first argument
        | CALL_START
        | CALL_START_SPACES
        ;

indent_option:
          '(' pos_type ',' NUMBER ')'
        ;

style_rule_list:
          style_rule_list style_rule
        | style_rule
        ;

style_rule :
          // A Compilation unit
          CODE_FILE  '{' file_rule_list '}'
          // Preprocessor
        | PP         '{' pp_rule_list '}'
          // Function definition
        | FUNC_DEF   '{' func_def_rule_list '}'
          // Function declaration
        | FUNC_DECL  '{' func_decl_rule_list '}'
          // Executable statements
        | STATEMENT  '{' stmnt_rule_list '}'
        | COMMENT    '{' comnt_rule_list '}'
          // Literal and string constants
        | CONSTANT   '{' const_rule_list '}'
        | EXPRESSION '{' expr_rule_list '}'
          // Data definition and declaration statements
        | DDD        '{' ddd_rule_list '}'
          // Operator
        | OP     .   '{' op_rule_list '}'
          // Identifier names
        | NAME       '{' name_rule_list '}'
        ;
```

```
file_rule_list:
          file_rule_list ';' file_rule
        | file_rule
        ;

file_rule:
          COMMENT '{' file_comnt_rule_list '}'
        | SIZE    '{' file_size_rule_list '}'
          // End of file shall be commented
        | COMMENT_EOF '=' mask
          // Comment every block of code
        | COMMENT_BLOCK '=' mask
          // Min size of block which shall be commented
        | COMMENT_BLOCK_SIZE '=' NUMBER
          // File shall have a header comment
        | HEADER '=' mask
          // Backslash newline pairse only in macro definitions
        | BS_NL '=' mask
          /* Terminate via exit  */
        | NO_RETURN '=' mask
          /* Exit operand shall be OS define value */
        | EXIT_TO_OS '=' mask
          /* Idempotent file includes */
        | IF_INCLUDE '=' mask
        | LINT_WARNINGS '=' mask
        | REDEFINE_STD_NAMES '=' mask
          // Type of info file header shall have
        | HEADER_FORMAT '=' '(' string_list ')'
        | TABWIDTH '=' NUMBER
        | INDENT_WIDTH '=' NUMBER
          // Maximum lines in a file
        | MAX_SIZE '=' NUMBER
          // Number of blank lines separating two function definitions
        | FUNC_SEPARATOR '='  NUMBER
        | BLOCK_SEPARATOR '=' NUMBER
          // Max number of non blank lines
        | MAX_BLOCK_SIZE '=' NUMBER
```

```
          //
          // Order of statements in a compilation unit.
          // Order value shall be defined for all statement types.
          // If order is not specified of some statement type then,
          // occurance of that type of statement will be considered as
          // a style violation.
          //
        | ORDER   '=' '(' stmnt_order ')'
        ;

file_comnt_rule_list:
          file_comnt_rule_list ';' file_comnt_rule
        | file_comnt_rule
        ;

file_comnt_rule :
        | END '=' mask
        | BLOCK '=' mask
        | HEADER '=' mask
        | ALL '=' mask
        ;

file_size_rule_list :
          file_size_rule_list ';' file_size_rule
        | file_size_rule
        ;

file_size_rule :
          CODE_FILE '=' NUMBER
        | BLOCK '=' NUMBER
        | INDENT '=' NUMBER
        | TABWIDTH '=' NUMBER
        | FUNC_SEPARATOR '='  NUMBER
        | BLOCK_SEPARATOR '=' NUMBER
        ;

stmnt_order :
          stmnt_order ',' '(' stmnt_type ',' NUMBER ')'
```

```
        | '(' stmnt_type ',' NUMBER ')'
        ;

stmnt_type:
          // System include
          SYS_INCLUDE
          // User defined include
        | USER_INCLUDE
          // Macro constant
        | MACRO_CONST
          // Macro function
        | MACRO_FUNC
        | TYPEDEF
          // struct/union definition
        | STRUCT
          // enum definition
        | ENUM
          // extern identifier declaration
        | EXTERN_VAR
          // extern function definition
        | EXTERN_FUNC
          // static function declaration
        | STATIC_FUNC_DECL
          // static identifier definition
        | STATIC_VAR
          // Global identifier definition
        | GLOBAL_VAR
          // Function definition
        | FUNC_DEF
          // static function definition
        | STATIC_FUNC_DEF
        ;

func_decl_rule_list:
          func_decl_rule_list ';' func_decl_rule
        | func_decl_rule
        ;
```

```
func_decl_rule:
          POSITION '{' func_decl_pos_rule_list '}'
          // Define argument names and types
        | DEF_ARG_NAMES '=' mask
        | MAX_ARGS '=' NUMBER
        | MAX_ARGS_PER_LINE '=' NUMBER
        | START_POS '=' indent_option
        | NAME_POS '=' indent_option
        | ARGS_START_POS '=' indent_option
          // Continuation position of arguments
        | ARGS_CONT_POS '=' indent_option
          // Disallowed return types
        | BAD_RET_TYPES '=' '(' type_list ')'
          // Specify return type
        | DEF_RET_TYPE '=' mask
          // Spaces between function name and open parenthesis
        | BIND_SPACES '=' indent_option
        | REDEFINE_NAMES '=' mask
          // Redefine standard names
        | REDEFINE_STD_NAMES '=' mask
          // Use variable length arguments
        | USE_VARARGS '=' mask
        ;

type_list:
          type_list ',' type
        | type
        ;

type:
          VOID
        | CHAR
        | UCHAR          // unsigned char
        | SHORT
        | USHORT        // unsigned short
        | INT
        | UINT           // unsigned int
        | LONG
```

```
        | ULONG          // unsigned long
        | FLOAT
        | DOUBLE
        | STRUCT
        | UNION
        | ENUM
        | USERTYPE   // typedefd type
        | ARRAY
        | PTR              // pointer
        ;

func_decl_pos_rule_list:
          func_decl_pos_rule_list ';' func_decl_pos_rule
        | func_decl_pos_rule
        ;

func_decl_pos_rule:
          ARGS_START '=' indent_option
        | ARGS_CONT  '=' indent_option
        | START '=' indent_option
        | NAME '=' indent_option
        ;

func_def_rule_list:
          func_def_rule_list ';' func_def_rule
        | func_def_rule
        ;

func_def_rule:
          POSITION '{' func_def_pos_rule_list '}'
        | SIZE     '{' func_def_size_rule_list '}'
        | COMMENT  '{' func_def_comnt_rule_list '}'

          // Data declaration/definition rules
        | DDD      '{' func_def_ddd_rule_list '}'
          // Prefix string for definition
        | PREFIX '=' STRING
        | MAX_ARGS '=' NUMBER
```

```
| USE_ENVP '=' mask
| DECL_EXTERN_VARS '=' mask
| DECL_EXTERN_FUNCS '=' mask
  // Define struct/union
| DEF_STRUCT '=' mask
| DEF_ENUM '=' mask
| DEF_TYPEDEF '=' mask
| DEF_STATIC_VARS '=' mask
| REDEFINE_NAMES '=' mask
| REDEFINE_STD_NAMES '=' mask
| BAD_RET_TYPES '=' '(' type_list ')'
| DEF_RET_TYPE '=' mask
| SINGLE_RETURN '=' mask
| END_COMMENT '=' mask
| END_COMMENT_BODY_SIZE '=' NUMBER
  /* Function header comment  */
| COMMENT_HEADER '=' mask
| COMMENT_ARGS '=' mask
| COMMENT_STATIC_VARS '=' mask
| COMMENT_BLOCK '=' mask
| MAX_SIZE '=' NUMBER
| MAX_BLOCK_SIZE '=' NUMBER
| DECL_SEPARATOR_SIZE '=' NUMBER
| BLOCK_SEPARATOR_SIZE '=' NUMBER
| MAX_ARGS_PER_LINE '=' NUMBER
| ARGS_START_POS '=' indent_option
| ARGS_CONT_POS '=' indent_option
  // Start column of return type
| START_POS '=' indent_option
| NAME_POS '=' indent_option
| OPEN_BRACE_POS '=' indent_option
| CLOSE_BRACE_POS '=' indent_option
  // Start column of local declarations/definitions
| DECL_START_POS '=' indent_option
| STATEMENT_POS '=' indent_option
| BIND_SPACES '=' indent_option
| USE_VARARGS '=' mask
;
```

```
func_def_pos_rule_list:
          func_def_pos_rule_list ';' func_def_pos_rule
        | func_def_pos_rule
        ;

func_def_pos_rule:
          // Start column of return type
          START '=' indent_option
        | ARGS_START '=' indent_option
        | ARGS_CONT '=' indent_option
        | NAME '=' indent_option
        | OPEN_BRACE '=' indent_option
        | CLOSE_BRACE '=' indent_option
        | DECL_START '=' indent_option
        | STATEMENT_POS '=' indent_option
        ;

func_def_size_rule_list:
          func_def_size_rule_list ';' func_def_size_rule
        | func_def_size_rule
        ;

func_def_size_rule:
          FUNC '=' NUMBER
        | BLOCK '=' NUMBER
        | DECL_SEPARATOR '=' NUMBER
        | BLOCK_SEPARATOR '=' NUMBER
        ;

func_def_comnt_rule_list:
          func_def_comnt_rule_list ';' func_def_comnt_rule
        | func_def_comnt_rule
        ;

func_def_comnt_rule:
          ALL '=' mask
        | END '=' mask
```

```
        | HEADER '=' mask
        | BLOCK '=' mask
        | STATIC_VAR '=' mask
        | ARGS '=' mask
        ;

func_def_ddd_rule_list:
          func_def_ddd_rule_list ';' func_def_ddd_rule
        | func_def_ddd_rule
        ;

func_def_ddd_rule:
          EXTERN_VAR '=' mask
        | EXTERN_FUNC '=' mask
        | RET_TYPE '=' mask
        | STRUCT '=' mask
        | ENUM '=' mask
        | TYPEDEF '=' mask
        | STATIC_VAR '=' mask
          // Variable number of arguments
        | VARARGS '=' mask
          // Sets every thing in the above list except return type
        | ALL '=' mask
         ;

pp_rule_list:
          pp_rule_list ';' pp_rule
        | pp_rule
        ;

pp_rule:
          MACRO_CONST '{' macro_const_rule_list '}'
        | MACRO_FUNC  '{' macro_func_rule_list '}'
        | INCLUDE      '{' include_rule_list '}'
          // #if, #ifdef, #ifndef
        | PP_IF        '{' pp_if_rule_list '}'
          // Rules pertaining to # and directive name
        | HASH         '{' hash_rule_list '}'
```

```
        ;

hash_rule_list:
          hash_rule_list ';' hash_rule
        | hash_rule
        ;

hash_rule:
          HASH_POS '=' indent_option
        | DIRECTIVE_POS '=' indent_option
        ;

include_rule_list:
          include_rule_list ';' include_rule
        | include_rule
        ;

include_rule:
        | COMMENT '=' mask
        | RELATIVE_FILE_NAME '=' mask
        | MAX_NESTED_INCLUDE '=' NUMBER
        ;

macro_const_rule_list:
          macro_const_rule_list ';' macro_const_rule
        | macro_const_rule
        ;

macro_const_rule:
        | EMITATES_TYPEDEF '=' mask
        | SEMICOLON_AT_END '=' mask
        | ASSIGN_AT_START '=' mask
        | BRACKETED_DEF '=' mask
          // Redefine macros
        | REDEFINE '=' mask
        | COMMENT '=' mask
          // Size in number of lines
        | MAX_SIZE '=' NUMBER
```

```
        ;

macro_func_rule_list:
          macro_func_rule_list ';' macro_func_rule
        | macro_func_rule
        ;

macro_func_rule:
          // Arguments in replacement text shall in parenthesis
        | BRACKETED_ARGS '=' mask
          // Replacement text shall in braces/parenthesis
        | BRACKETED_DEF '=' mask
        | SEMICOLON_AT_END '=' mask
        | MIN_ARGS '=' NUMBER
        | MAX_ARGS '=' NUMBER
          // Redefine macro functions
        | REDEFINE '=' mask
        | COMMENT '=' mask
          // Position of baclslash
        | BS_NL_POS '=' indent_option
        | CONT_POS '=' indent_option
        | START_POS '=' indent_option
        | ARGS_START_POS '=' indent_option
        | ARGS_CONT_POS '=' indent_option
        ;

pp_if_rule_list:
          pp_if_rule_list ';' pp_if_rule
        | pp_if_rule
        ;

pp_if_rule:
          // use preprocessor's defined operator
        | USE_DEFINED '=' mask
          // comment #else and #elseif
        | COMMENT_ELSE '=' mask
        | COMMENT_ENDIF '=' mask
          // Min body size of #if, #else and #elseif to have comment
```

```
        | ENDIF_COMMENT_BODY_SIZE '=' NUMBER
        | MAX_NESTED_IF '=' NUMBER
        ;

const_rule_list:
          const_rule_list ';' const_rule
        | const_rule
        ;

const_rule:
          // Use only cpital letters in literal costants
          CAPITALIZE '=' mask
          // Use only decimal constants
        | DECIMALS '=' mask
          // use hexa-decimal constants in bitwise operations
        | HEX_IN_BIT_STATEMENTS '=' mask
          // All character constants shall be #defined
        | DEFINED_CHAR_CONSTS '=' mask
          // List of magic constants which can be used as literal constants
        | MAGIC_LIST '=' '(' string_list ')'
          // Use magic constants
        | MAGIC_CONSTS '=' mask
        ;

string_list:
          string_list ',' STRING
        | STRING
        ;

stmnt_rule_list:
          stmnt_rule_list ';' stmnt_rule
        | stmnt_rule
        ;

stmnt_rule:
          IF      '{' if_rule_list '}'
        | SWITCH  '{' switch_rule_list '}'
          // for, do and while loops
```

```
| LOOP    '{' loop_rule_list '}'
| COMMENT '{' stmnt_comnt_rule_list '}'
| STATEMENTS_PER_LINE '=' NUMBER
| USE_GOTO '=' mask
| COMMENT_GOTO '=' mask
  // spaces between if/swicth/for/while and following parenthesis
| BIND_SPACES '=' indent_option
  // spaces between return and following parenthesis
| RETURN BIND_SPACES '=' indent_option
  // Put return argument in parenthesis
| BRACKETED_RETURN '=' mask
  // use embeded assignments
| MULTIPLE_ASSIGNS '=' mask
| LABEL_START_POS '=' indent_option
  // label shall be on a line by itself.
| EMPTY_LABEL_LINE '=' mask
| COMMENT_NON_PORTABLE '=' mask
| COMMENT_BIT_STATEMENTS '=' mask
| NULL_IS_PTR '=' mask
  // use default type promotion/typecast rules
| DEFAULT_TYPECAST '=' mask
  // typecast pointer other than to/from void or signed and
  // unsigned of same types
| TYPECAST_PTR '=' mask
| MULTIPLE_FUNC_CALLS '=' mask
  // Assign higher range vars to lower range vars.
| CHECK_RANGE '=' mask
  // use negative number in integer division
| NEGATIVE_INT_DIV '=' mask
| NULL_NOT_EOS '=' mask
| CHECK_SYS_CALL '=' mask
  // check call memory allocation functions
| CHECK_MEM_CALL '=' mask
| BRACED_STATEMENTS '=' mask
  // if, switch, for, do, while shall have a
  // comment at end of their body.
| END_COMMENT '=' mask
| END_COMMENT_BODY_SIZE '=' mask
```

```
        | OPEN_BRACE_POS '=' indent_option
        | CLOSE_BRACE_POS '=' indent_option
        | NULL_BODY_POS '=' indent_option
        | COMMENT_NULL_BODY '=' mask
        | MAX_SIZE '=' NUMBER
        ;

stmnt_comnt_rule_list:
          GOTO '=' mask
        | NON_PORTABLE '=' mask
        | BIT_STATEMENTS '=' mask
        | END '=' mask
        | NULL_BODY '=' mask
        ;

switch_rule_list:
          switch_rule_list ';' switch_rule
        | switch_rule
        ;

switch_rule:
          // use fall through feature
        | FALL_THROUGH '=' mask
        | COMMNT_FALL_THROUGH '=' mask
          // Body in braces always
        | BRACED '=' mask
        | OPEN_BRACE_POS '=' mask
        | CLOSE_BRACE_POS '=' mask
        | END_COMMENT '=' mask
        | END_COMMENT_BODY_SIZE '=' mask
        | MAX_SIZE '=' NUMBER
        | BIND_SPACES '=' indent_option
          // use default always
        | DAFAULT_CASE '=' mask
          // default shall be the last condition
        | DEFAULT_CASE_LAST '=' mask
          // Body case shall be be braces
        | BRACED_CASE '=' mask
```

```
        // Max number of lines of case body
      | MAX_CASE_SIZE '=' mask
      | CASE_POS '=' mask
      | STATEMENT_POS '=' mask
        // case condition shall be on a line by itself
      | EMPTY_CASE_LINE '=' mask
        // number of case conditions per line
      | CASES_PER_LINE '=' NUMBER
      | COMMENT_CASE '=' mask
      ;

loop_rule_list:
        loop_rule_list ';' loop_rule
      | loop_rule
      ;

loop_rule:
        // Body shall always be in braces
        BRACED '=' mask
      | END_COMMENT '=' mask
      | END_COMMENT_BODY_SIZE '=' NUMBER
      | MAX_SIZE '=' NUMBER
      | COMMENT_BREAK '=' mask
      | COMMENT_CONTINUE '=' mask
      | OPEN_BRACE_POS '=' indent_option
      | CLOSE_BRACE_POS '=' indent_option
      | NULL_BODY_POS '=' indent_option
      | COMMENT_NULL_BODY '=' mask
      | BIND_SPACES '=' indent_option
      | STATEMENT_POS '=' indent_option
        // max length in chars of for's initial;ization, condition and
        // iteration statement. If expr is lengthier than it the
        // it shold be split over multiple lines.
      | MAX_FOR_EXPR_LENGTH '=' NUMBER
      ;

if_rule_list:
        if_rule_list ';' if_rule
```

```
        | if_rule
        ;

if_rule:
          BRACED '=' mask
        | END_COMMENT '=' mask
        | END_COMMENT_BODY_SIZE '=' NUMBER
        | MAX_SIZE '=' NUMBER
        | OPEN_BRACE_POS '=' indent_option
        | CLOSE_BRACE_POS '=' indent_option
        | NULL_BODY_POS '=' indent_option
        | COMMENT_NULL_BODY '=' mask
        | STATEMENT_POS '=' indent_option
        | BIND_SPACES '=' indent_option
        | BRACED_NESTED_IF '=' mask
        | ELSE_IF_SPACES '=' indent_option
        ;

expr_rule_list:
          expr_rule_list ';' expr_rule
        | expr_rule
        ;

expr_rule:
          INDEX '{' index_expr_rule_list '}'
        | LOGIC '{' logic_expr_rule_list '}'
        | CALL  '{' call_expr_rule_list '}'

          // long/multiline expressions shall be split after op/operand
        | SPLIT_AFTER '=' OPERAND
        | SPLIT_AFTER '=' OP
        | CONT_POS '=' indent_option
          // Valid operators with pointer type variables
        | PTR_OPS '=' '(' op_list ')'
          // ptr shall be compared with NULL or ids of identical ptr types
        | NULL_PTR_COMPARE '=' mask
          // EOF shall be compared with signed type only
        | SIGNED_EOF_COMPARE '=' mask
```

```
          // max operators per line
        | MAX_OPS_PER_LINE '=' NUMBER
          // max operators per expression
        | MAX_OPS_PER_EXPR '=' NUMBER
        ;

op_list:
          op_list ',' op_code
        | op_code
        ;

index_expr_rule_list:
          index_expr_rule_list ';' index_expr_rule
        | index_expr_rule
        ;

index_expr_rule:
          BAD_OPS '=' '(' op_list ')'
        | SPLIT_AFTER '=' OP
        | SPLIT_AFTER '=' OPERAND
        | CONT_POS '=' indent_option
        | BRACKETED '=' mask
        ;

logic_expr_rule_list:
          logic_expr_rule_list ';' logic_expr_rule
        | logic_expr_rule
        ;

logic_expr_rule:
          BAD_OPS '=' '(' op_list ')'
        | SPLIT_AFTER '=' OP
        | SPLIT_AFTER '=' OPERAND
        | CONT_POS '=' indent_option
          // conditional expression must have at least one relational op
        | DEFAULT_FAIL '=' mask
        | BRACKETED '=' mask
        ;
```

```
call_expr_rule_list:
          call_expr_rule_list ';' call_expr_rule
        | call_expr_rule
        ;

call_expr_rule:
          SPLIT_AFTER '=' NUMBER
        | CONT_POS '=' indent_option
          // Spaces between function name and parenthesis in function call
        | BIND_SPACES '=' indent_option
          // Disallowed ops in call expression
        | BAD_OPS '=' '(' op_list ')'
        | MACRO_CALL_BAD_OPS '=' '(' op_list ')'
        | CAST_NULL '=' mask
        | CAST_ARGS '=' mask
        ;

comnt_rule_list:
          comnt_rule_list ';' comnt_rule
        | comnt_rule
        ;

comnt_rule:
          BLOCK '{' block_comnt_rule_list '}'
        | START_SPACES '=' NUMBER
        | END_SPACES '=' NUMBER
        | START_CASE '=' case_option
        | TEXT_CASE_OPTION '=' case_option
        | INDENT_TEXT '=' mask
        ;

block_comnt_rule_list:
          block_comnt_rule_list ';' block_comnt_rule
        | block_comnt_rule
        ;

block_comnt_rule:
```

```
        START_POS '=' indent_option
      | TEXT_START_POS '=' indent_option
      | CONT_POS '=' indent_option
      | END_POS '=' indent_option
      | PREFIX_STRING '=' STRING
      | SUFFIX_STRING '=' STRING
      ;

ddd_rule_list:
        ddd_rule_list ';' ddd_rule
      | ddd_rule
      ;

ddd_rule:
        STRUCT '{' struct_rule_list '}'
      | ARRAY  '{' array_rule_list '}'
      | ENUM   '{' enum_rule_list '}'

        // max lines in decl/definition statement
      | MAX_LINES '=' NUMBER
      | MAX_VARS_PER_LINE '=' NUMBER
        // initialize identifers
      | INIT_VARS '=' mask
      | INIT_STATIC_VARS '=' mask
        // Every decl/definition shall have one type of void, char,
        // short, int, long, float or double.
      | DEFAULT_INT_TYPE '=' mask
      | COMMENT_GLOBAL_VARS '=' mask
      | COMMENT_STATIC_VARS '=' mask
        // comment const declaations/definitions
      | COMMENT_CONST '=' mask
      | START_POS '=' indent_option
      | CONT_POS '=' indent_option
        // Portable #define types shall be used.
      | PORTABLE_TYPES '=' mask
        // All typedefs shall be defined in a separate file
      | TYPEDEF_FILE '=' mask
        // All struct/unions shall be defined in a separate
```

```
        | STRUCT_FILE '=' mask
          // Number spaces between * and id in pointer decl/definition
        | PTR_BIND_SPACES '=' indent_option
        ;

enum_rule_list:
          enum_rule_list ';' enum_rule
        | enum_rule
        ;

enum_rule:
          ENUM_BOOLEAN '=' mask
        | COMMENT '=' mask
        | COMMENT_CONST '=' mask
        | MAX_CONSTS_PER_LINE '=' NUMBER
          // typename and enum shall be defined together
        | MIX_WITH_TYPEDEF '=' mask
          // enum definition shall not be combined with var definitions.
        | MIX_WITH_VARS '=' mask
          // enum definition shall not be combined with function
          // definitions/declaration.
        | MIX_WITH_FUNC '=' mask
        | INIT_CONSTS '=' mask
        | DEFINE_TAG '=' mask
        | OPEN_BRACE_POS '=' indent_option
        | CLOSE_BRACE_POS '=' indent_option
        | CONST_START_POS '=' indent_option
        | CONST_CONT_POS '=' indent_option
        ;

struct_rule_list:
          struct_rule_list ';' struct_rule
        | struct_rule
        ;

struct_rule:
          MIX_WITH_TYPEDEF '=' mask
        | MIX_WITH_VARS '=' mask
```

```
        // Combine struct/union definition with function
        // declaration/definition.
      | MIX_WITH_FUNC '=' mask
        // typedef for struct/union shall have typename for struct and
        // struct ptr.
      | TYPEDEF_PTR '=' mask
        // union shall not have pointer fields combined with non
        // pointer type fields
      | PTR_ONLY_UNION '=' mask
      | COMMENT '=' mask
      | COMMENT_FIELDS '=' mask
      | COMMENT_BITFIELDS '=' mask
      | UNSIGN_BITFIELDS '=' mask
      | STRUCT_FILE '=' mask
      | FIELDS_PER_DECL '=' NUMBER
      | DEFINE_TAG '=' mask
      | OPEN_BRACE_POS '=' indent_option
      | CLOSE_BRACE_POS '=' indent_option
      | FIELDS_START_POS '=' indent_option
      | MAX_FILEDS = NUMBER
      ;


array_rule_list:
        array_rule_list ';' array_rule
      | array_rule
      ;

dim_list:
        dim_list ',' NUMBER
      | NUMBER
      ;

array_rule:
        // Most significant dimension of external array shall not be
        // defined.
        NULL_FIRST_DIM '=' mask
        // Most significant dimension of initialized array shall
```

```
          // not be defined.
        | INIT_FIRST_DIM '=' mask
          // Valid ops to be used with array vars.
        | VALID_OPS '=' '(' op_list ')'
          // Dimension expression shall be a #defined expression
        | MAGIC_DIMS '=' mask
        | MAX_DIM_SIZE '=' '(' dim_list ')'
        ;

op_rule_list:
          op_rule_list ';' op_rule
        | op_rule
        ;

op_rule:
          // Conditional expression of ternary operator shall inclosed
          // in parenthesis
          BRACKETED_TERNARY '=' mask
          // Nested ternary ops shall not be used.
        | NESTED_TERNARY '=' mask
          // Relational ops etc shall be #defined
        | NAMED_OPS '=' '(' op_name_list ')'
          // Spaces around a op
        | OP_SPACES '=' '(' op_space_list ')'
        | ARRAY_FORMAT '=' NUMBER '[' NUMBER ']' NUMBER
          // Number of spaces after a comma.
        | COMMA_SUFFIX_SPACES '=' NUMBER
          // Number of spaces after a semi-colon.
        | SEMICOLON_SUFFIX_SPACES '=' NUMBER
        | BRACKETED_SIZEOF '=' mask
          // Braces shall be on a line by itself
        | EMPTY_BRACE_LINE '=' mask
        ;

op_name_list:
          op_name_list ',' op_name
        | op_name
        ;
```

```
op_name:
          '(' op_code '=' STRING ')'
        ;

op_space_list:
          op_space_list ',' op_space
        | op_space
        ;

op_space:
          '(' op_code '=' NUMBER ',' NUMBER ')'
        ;

op_code:
          PRIMARY
        | BINARY
        | UNARY
        | ASSIGN
        | CALL
        | INDEX
        | ARROW
        | '.'
        | LNOT                  // Logical not
        | '~'
        | PLUSPLUS
        | MINUSMINUS
        | UMINUS                // unary minus
        | UPLUS                 // unary plus
        | STAR                  // dereference
        | ADDRESS               //  &
        | CAST
        | SIZEOF
        | '*'                   // multiplication
        | '/'
        | '%'
        | '+'
        | '-'
```

```
    // Max length of an external variable.
  | MAX_EXTERN_VAR_LENGTH = NUMBER
  | MAX_FILE_NAME_LENGTH '=' NUMBER
    // full_length . length_of_extension
  | FILE_NAME_FORMAT '=' NUMBER '.' NUMBER
  | MIN_NAME_LENGTH '=' NUMBER
    // Text case of const type id.
  | CONST_CASE '=' case_option
    // Text case of enum constants
  | ENUM_CASE '=' case_option
    // Text case of #defined constants
  | MACRO_CONST_CASE '=' case_option
    // Text case of #defined constants
  | MACRO_FUNC_CASE '=' case_option
    // Text case of variable names
  | VARNAME_CASE '=' case_option
    // Text case of typedefed type names.
  | TYPENAME_CASE '=' case_option
    // Text case of function names
  | FUNCNAME_CASE '=' case_option
    // Text case of label names.
  | LABEL_NAME_CASE '=' case_option
  | STRUCT_TAG_CASE '=' case_option
  | ENUM_TAG_CASE '=' case_option
    // Max length subword. _ / change of case is considered
    // as end of a subword.
  | MAX_SUB_WORD_LENGTH '=' NUMBER
  | NAMING_STYLE '=' CASE_CHANGE
  | NAMING_STYLE '=' UNDERSCORE
  | PREFIX_FUNCNAME '=' mask
  | PREFIX_ARGNAMES '=' mask
  | TYPE_PREFIXS '=' '(' prefix_list ')'
  | UNDERSCORE_START '=' mask
  ;

prefix_list:
    prefix_list ',' type_prefix
  | type_prefix
```

```
        ;

type_prefix:
          '(' type '=' STRING ')'
        ;
```

Appendix B

Sample Style File

```
CODE_FILE
{
    HEADER = YES; // File shall have a header comment
    BS_NL = NO; // back slash new-lines only in macros
    NO_RETURN = YES; // main shall return with exit call
    EXIT_TO_OS = YES; // exit arg shall be OS defined value
    IF_INCLUDE = YES // idempotent file includes
}
FUNC_DEF
{
    MAX_ARGS_PER_LINE = 1;
    COMMENT_HEADER = YES; // function shall have a header comment
    DECL_EXTERN_VARS = NO; // extern var decl shall not be there
    USE_ENVP = NO; // envp command line arg shall not be used
    DEF_STRUCT = NO; // struct/union shall not be defined
    DEF_RET_TYPE = NO; // return type shall be specified
    SINGLE_RETURN = NO; // function shall have a single exit point
    USE_VARARGS = NO // variable length args shall not be used
}
PP
{
    INCLUDE
    {
        RELATIVE_FILE_NAME = YES
    };
```

```
    MACRO_FUNC
    {
        BRACKETED_ARGS = YES; // args shall be in parenthesis
        SEMICOLON_AT_END = NO
    }
}
CONSTANT
{
    HEX_IN_BIT_STATEMENTS = YES;
    DECIMALS = YES // only decimal constants shall be used
}

OP
{
    COMMA_SUFFIX_SPACES = 1;
    NAMED_OPS = ( (== ''EQUAL'') );
}
STATEMENT
{
    IF
    {  // nested if shall be braced completely
        BRACED_NESTED_IF = YES;
    };
    USE_GOTO = NO;
    NULL_IS_PTR = YES; // NULL shall be used as a pointer
    MULTIPLE_FUNC_CALLS = NO; // Only one function call
    CHECK_SYS_CALL = YES;
    CHECK_MEM_CALL = YES;
    END_COMMENT = YES;
    NULL_BODY_POS = (ALIGN, 1);
    DEFAULT_CASE = YES; // switch shall have a default case
}
EXPRESSION
{
    PTR_OPS = (+, -); // ops to be used with ptr type vars
    SIGNED_EOF_COMPARE = YES; // EOF shall be compared with signed type
    CALL
    {
```

```
        BIND_SPACES = (SPACES, 1);
        MACRO_CALL_BAD_OPS = (++, --) // ops not to used
    };
    LOGIC
    {
        BAD_OPS = (=, +=, -=, &=, !=, /=, ^=, <<=, >>=, %=);
        DEFAULT_FAIL = NO // conditional expression shall have
                          // at least on relational op
    }
}
DDD
{
    COMMENT_STATIC_VARS = YES;
    DEFAULT_INT_TYPE = NO;
    MAX_VARS_PER_LINE = 1;
    MAX_LINES = 1;
    STRUCT_FILE = YES; // struct definitions shall be kept in a separate file
    STRUCT
    {
        MIX_WITH_TYPEDEF = YES; // typedef of struct shall be combined
                                // with struct definition
        MIX_WITH_VARS = NO
    };
    ARRAY
    {
        MAGIC_DIMS = NO // dimension expr shall be #defined expr
    }
}
NAME
{
    NAMING_STYLE = UNDERSCORE;
    CONST_CASE = UPPER;
    FILE_NAME_FORMAT = 11.3;
    PREFIX_ARGNAMES = YES;
    UNDERSCORE_START = NO
}
```

Appendix C

Sample Style Violation Warnings

```
CSA Warning:test/t1.c:1:Missing file header
CSA Warning:test/t1.c:5:macro constant name in improper case
CSA Warning:test/t1.c:9:UnCommented global static var
CSA Warning:test/t1.c:11:UnCommented global var
CSA Warning:test/t1.c:15:enum constant name in improper case
CSA Warning:test/t1.c:15:Insufficient spaces after ,
CSA Warning:test/t1.c:24:typename in improper case
CSA Warning:test/t1.c:33:Function name starts from wrong position
CSA Warning:test/t1.c:36:Function declaration inside function
CSA Warning:test/t1.c:38:Uncommented local static var
CSA Warning:test/t1.c:39:Var definition spans over lines
CSA Warning:test/t1.c:39:More number of definition in a single line
CSA Warning:test/t1.c:53:More number of definition in a single line
CSA Warning:test/t1.c:59:Insufficient spaces after if
CSA Warning:test/t1.c:65:Invalid op = in conditional expression
CSA Warning:test/t1.c:65:else if position invalid
CSA Warning:test/t1.c:60:No braces around nested if statement
CSA Warning:test/t1.c:69:Insufficient spaces after for
CSA Warning:test/t1.c:74:Insufficient spaces after while
```

Appendix D

Glossary

ARG : argument
BS_NL : backslash newline pair
C-CSC : C Coding Style Checking
C-SSL : C Style Specification Language
CALL : call expression
CONST : constant
CONT : continuation
DDD : Data declaration and definition
DECL : declaration
DEF : definition
EOF : end of file
EXPR : expression
FUNC : function
GNU : Generally not unix
INDEX : index expression
INIT : initialize
LOGIC : logic/conditional expression
OP : operator
OS : Operating system
POS : position

PP : preprocessor

PTR : pointer

RET : return

STD : standard

SYS : system

VAR : variable/identifier

References

[AD90] A. Lemmke etc. A. Dolenc. Notes on writing portable programs in C. Technical report, Internet Document, Nov 90. ftp://cs.washington.edu/pub/cport.tar.Z.

[Ara92] Mouloud Arab. Enhancing program comprehension: Formatting and documentation. *ACM SIGPLAN Notices*, 27(2):37, Feb 92.

[Arc94] Joseph Arceneaux. *Indent.* GNU, 1.3 edition, Jan 94.

[AT 92] AT & T. *Unix System V Release 4, Programmer's Guide, ANSI C & Programming Support Tools*, 92.

[Cod90] Three low cost code analyzers. Sept 90.

[DC85] Ian Darwin and Geoff Collyer. Can't happen or /* NOTREACHED */ or real programs dump core. *USENIX*, Jan 85.

[DE95] Prem Devanbu and Laura Eaves. Gen++ an analyzer generator for C++ programs. Technical report, AT and T, 1995. A Technical report.

[DEH94] John Guttag David Evans and James Horning. LClint:a tool for using specifications to check code. *Software Engineering Notes*, page 87, Dec 94.

[EE85] Berry R. E. and Meeking B. A. E. A style analysis of C programs. *Communications of ACM*, 28(1):80, Jan 85.

[faq96] C-faq, Feb 96. Available at ftp.eslimo.com.

[FSF94] FSF. GNU C coding standard. Technical report, Internet Document, Dec 94.

[HS89] Keppel D. and Spencer H. Recommended C style and coding standard. *Internet Document*, Nov 89. ftp://cs.washington.edu/pub/cstyle/cstyle.tex.

[J.85] Arthur L. J. *Measuring Programmer's Productivity and Software Quality.* John Willey, 85.

[Koe89] Andrew Koenig. *C Traps and Pitfalls.* Addison Wesley, 89.

[KR90] Brian W. Kernighan and Dennis M. Rithche. *The C Programming Language.* Prentice Hall, 2nd edition, July 90.

[OC88] Paul W. Oman and Curlis R. Cook. A paradigm of programming style research. *ACM SIGPLAN Notices*, Dec 88.

[Ous89] John K Ousterhout. *Sprite Engineering Manual.* University of California, May 89.

[Pik90] Rob Pike. Notes on programming in C. Technical report, Internet Document, 90. ftp://cs.washington.edu/pub/cstyle/pikestyle.ms.

[Pre88] Roger S. Pressman. *Software Engineering - A Practitioners Approach.* McGraw Hill, 2nd edition, 88.

[RS86] K. A. Redish and W. F. Smyth. Program style analysis: A natural by-product of program compilation. *Communications of ACM*, page 126, Feb 86.

[Tas78] Dennie Van Tassel. *Program Style, Design, Efficiency, Debugging and Testing.* Prentice Hall, 2nd edition, 78.

[Wal91] Michal Walicki. The stanford Ada style checker. Technical Report CSL-TR-91-488, Stanford University, Aug 91.

[WC89] Kirt A. Winter and Curtis R. Cook. A prototype intelligent prettyy printer for pascal. *ACM SIGPLAN Notices*, 25(9):116, Sept 89.